Создание графических приложений на платформе Qt. Краткий обзор (тезисы к семинару Томской группы пользователей Линукс).

Куликов Г.Э*

28 января 2006

Содержание

1 Мотивация **2**

2 Граф. подсистемы **5**

2.1 X – Window, стандартная граф. подсистема открытых систем 7

3 Графические тулкиты **8**

3.1 Беглый обзор. Что выбрать. 8

3.2 Как с этим работают 9

3.2.1 GTK 9

3.2.2 wxWidgets 10

3.2.3 Fox toolkit 10

3.3 QT 11

3.4 Общий обзор QT 12

3.4.1 Возможности 12

3.4.2 Реализация 13

3.4.3 Модель программирования 13

3.4.4 Локализация 14

3.4.5 Инструментарий 14

3.5 Базовые механизмы QT 14

3.5.1 Передача сообщений 14

3.5.2 Управление памятью 17

3.5.3 Навигация в иерархии 17

3.5.4 Поддержка многотредовости 17

3.5.5 Поддержка создания пользовательских виджетов 17

3.5.6 Особенности компиляции Qt приложений 18

*gleb@asd.iao.ru

List of Algorithms

1 скелет GTK приложения . . . 9
2 скелет wxWidgets приложения . . . 10
3 Описание класса, посылающего сигналы . . . 15

Список иллюстраций

1 графический интерфейс OS/2 PM . . . 5
2 классическая “визитка” X–Window . . . 6
3 Motif – приложения . . . 8
4 QSA Designer . . . 13

Об этом тексте

Этот текст — “рыба” доклада на самодеятельном семинаре Томской группы пользователей Линукс. Я старался оформить настоящий текст так, чтобы он был максмально полезным сам по себе, но поскольку планируется демонстрация работы в средах KDeveloper/QtDesigner, этот текст не может быть полным. Места, требующие дополнительной демонстрации, выделены значком $\otimes$ на полях.

1 Мотивация

Наверное, первый вопрос, приходящий в голову человеку, впервые попавшему на обсуждение продуктов открытых программных систем — “зачем мне это собственно, нужно? Зачем тратить силы и время на изучение (экотических) сред, особенно, если я — не программист и не техно–фанат, а пользователь, да ещё область профессиональных интересов которого далека от программистских заморочек? Да, частенько приходится, как программирующему пользователю, возиться с собственными программками, но ведь есть стандартные Win32API, MFC, Delphi”?

По моему мнению, говоря о использовании компьютеров и программных продуктов в повседневной деятельности, как рабочего инструмента, часто упускают из виду его универсальную природу. Если вы плотно используете компьютер, то быстро обнаруживаете, что даже если он был куплен как инструмент для решения одной единственной частной задачи — неважно, какой, ведения бухгалтерии или работы с конструкторскими чертежами, вы начинаете использовать машину как основное рабочее средство. В ЭВМ хранится всё! Ваша почтовая переписка, ксерокопии журнальных статей, вычислительный инструментарий — одним словом, всё. Тихо и незаметно, персональный компьютер из престижной игрушки конца 80-ых, стал основой всей современной цивилизации.

Но все знают, сам по себе компьютер, без наполняющих его программ — бесполезный кусок пластика и железа, не более. И хотя кажется, что всё что возможно, уже придумано, Вам постоянно требуются новые программы, новый “софт”, и более совершенные версии старых: меняется мир и мы меняемся вмести с ним. И “софт” разный — от серьёзных продуктов

известных фирм, до наколенных поделок собственного производства. Вам нужны средства взаимодействия с другими машинами, ваших коллег, друзей... Да попросту, простой инстинкт самосохранения (помните, почти вся ваша жизнь ныне хранится в этом сером ящике) требует, чтобы у вас была возможность в любой момент ясно и чётко понимать, что происходит на вашей машине и уверенность, что всё, что вы создаёте, может быть гарантировано воспроизведено на любом другом компьютере.

А что мы видим сейчас? С одной стороны, практически на всех машинах установлен однотипный “софт” (и даже более того, одного производителя — вы сами знаете эту фирму)... Но **что** при использовании этого программного обеспечения происходит на вашей машине — вы не знаете и более того, возобладал подход, что как любой домохозяйке, вам и не надо этого знать. Почти любой “документ (в широком сиысле — любой результат вашей деятельности)”, который вы создаёте — заметьте, **ваш** документ — де-юре, не может быть воспроизведён на машинах, где не установлен аналогичный софт того же производителя, поскольку способ, каким этот документ сохраняется – тщательно засекречен и разбираться в нём вам — по соглашению с производителем, которое вы подписали — не позволено. Это значит, что **ваш** документ — фактически — вам **не принадлежит**. Вы не можете сделать с ним ничего, что не предусмотрел производитель программного обеспечения. И производитель волен играть не вполне честно: для справки, документы, созданные в ранних версиях M$ Word, в современных часто не читаются, либо читаются неправильно. Другими словами, вы теряете доступ к вашему архиву и теряете только потому, что это выгодно производителю: вас вынуждают покупать новые вариации программного обеспечения, без всяких альтернатив. Вы не можете послать ваш документ любому коллеге, если у того также не куплен аналогичный набор программного обеспечения от всё того же производителя. Если вы сами создаёте программы для вашей повседневной деятельности, вы обязаны купить инструментарий всё у того же производителя, причём не факт, что вы сможете приобрести необходимый вам инструмент: цены устанавливаются произвольно и кое-что из “предметов первой необходимости” стоит неадекватно дорого, и уж совсем не факт, что вы сможете произвольным образом распорядиться результатами вашего труда: условия использования изготовленного вами “софта” оговариваются в договоре на купленные вами инструменты.

Для любых взаимотношений в обществе характерен обмен мыслями идеями, заимствование друг от друга удачных находок... вам запрещено вести себя таким образом тем же соглашением с производителями программного обеспечения. Вы не только не имеете права разбираться, как работает тот или иной программный продукт или почему он работает неправильно, часто вы не имеете права разрабатывать нечто новое, опираясь на купленный вами “софт”, или публиковать разработанное.

В то же время, цены на многие продукты устанавливаются совершенно произвольно, а набор гарантий, выдаваемых вам производителем в переводе с юридеческого языка на человеческий звучит как “вы обязаны купить наш продукт, вне не в праве посмотреть, как он работает, вы не можете вновь продать его, если он вам не подошёл, если из-за использования нашего продукта произошо что-то нехорошее — это ваши проблемы, мы вам ничего не должны”.

Картина вполне понятна: типичные отношения потребителя и монополиста, всеми силами стремящегося “подмять” по себя рынок. Разница только в том, что в результате ограничива-

ются не только ваши возможности, но и свобода — в самом широком смысле.

Пока мы не очень почувствовали на себе очевидные неприемлимые аспекты такого положения вещей, но по мере ужесточения контроля за распространением и использованием "контрафактного софта", будем всё чаще сталкиваться с самыми неприятными обстоятельствами.

Естественно, с таким положением дел мириться не стоит. Очевидно, что кроме софта коммерческого, требующего обязательного лицензирования, внутренние механизмы которого засекречены, должен существовать и софт общедоступный, допускающий открытое развитие. То, что сейчас называется Open Source.

Интересно, что принципы, действующие в научном сообществе и частично – в промышленном сообществе, в части общедоступных технологий, и принципы Open Source — близнецы. Кстати, открытость того, или иного представителя Open Source вовсе не обязательно означает его некоммерческую природу.

Общедоступный, публичный софт, развивается уже более 40 лет (вспомните старые публикации и сборники "алгоритмов и программ"). За это время, наработано огромное количество общедоступных и очень качественных программ, перекрывающих практически все области деятельности. Абсолютное большинство этих программ, кроме открытости, ещё и бесплатны и лицензионно – чисты.

Казалось бы, с точки зрения здравого смысла, разумно и финансово оправдано, в первую очередь, ориентироваться на окрытый софт, как общедоступный, и только потом — на софт коммерческий... но в реальности, ситуация совершенно противоположная. Частично это происходит из – за недостатка знаний и у пользователей, и у разрабочиков: последние 15 лет, весь процесс обучения стоится вокруг коммерческих и закрытых технологий.

Особенно обидно, когда просто не задумываются о том, что использовать: тот же Delphi, "купленый" за 50 рублей на блошином рынке, или легальные и открытые альтернативы. А когда приходит время публиковать результаты своей деятельности,оказывается, что сделать этого нельзя: ну сидел студент, делал на Delphi прибор... а в отчёте по гранту, зачем-то, оказалось необходимым подтвердить легальность использованого инструментария и прилетело в результате так, что мало не показалось никому...

Означает ли сказаное, что нет никаких серьёзных доводов использовать открытый, общедоступный, но в силу упомянтух причин, малознакомый софт вместо коммерческого, если вопросы легальности не стоят? Часто, открытое ПО проще в использовании и дешевле в эксплуатации. Возвращаясь к примеру с гипотетическим прибором и Delphi – софтом к нему: в выходящей вскоре новой версии Windows, Vista, вы не сможете самостоятельно создавать драйвера для вашего специфического "железа", не пройдя весьма недешёвого процесса сертификации и не получив в Майкрософт индивидуальный ключ цифровой подписи. Естественно, в случае открытого ПО, такой проблемы нет. А возможность гибкой конфигурации, позволяет использовать под многие типы приборов старые "персоналки", даже не имеющие винчестера, либо использовать одну и ту же программу как в нормальном, настольном компьютере, так и копеечном специализированном микроконтроллере.

Словом, говорить на эту тему можно долго. А пока моя цель — показать, что пользуясь открытым ПО можно создавать программы с современным графическим интерфейсом ничуть не сложнее и не медленнее, чем в любимом многими коммерческом Delphi.

Рис. 1: графический интерфейс OS/2 PM

2 Граф. подсистемы

Исторически, графические подсистемы вычислительных систем, с момента возникновения термина “графический интерфейс пользователя”, строились на двух идеологически разных принципах. С одной стороны, графический интерфейс впервые был разработан для интерактивной работы на машинах, которые сейчас относились бы к разряду персональных (компьютер Alto, Xerox PARC). Для этих машин, хотя и крайне совершенных для своего времени, остро стоял вопрос ресурсов и минимизации времени отклика на действия пользователя. Поэтому, развитие наследников концепции интерфейса Xerox (1980, Three Rivers Computer Corporation, “Perq”; 1983 Apple Lisa; 1984 – основы OS/2 PM; 1984, Apple Macintosh, 1985 M$ Windows) шло по пути построения всеобъемлющей и сложной библиотеки системных вызовов, по возможности, находящейся возможно ближе к ядру операционной системы. Пожалуй, наиболее ярким и совершенным представителем такой библиотеки (в части возможностей двумерной графики), до сих пор остаётся O2/2 PM (рис. 1), первая публичная версия которого вышла в свет только в 1988 году, через 4 года после начала разработки.

Для графических подсистем, пошедших по этому пути, характерны **1)** тесная интеграция с ОС и высокая сложность, обеспечивающая зато относительную простоту программирования на прикладном уровне, **2)** большое количество операций, идущих “за кулисами” системы, **3)** фактическая невозможность перенесения подсистемы, разработанной для одной операционной системы и типа процессора, в другую операционную и аппаратную среду. Кроме того, все персональные машины того времени, для которых велась разработка графических подсистем, были оборудованы сетевыми средствами скорее как исключение и ни в одной из первоначальных реализаций графики, за исключением OS/2 PM, не было сделано даже попытки обратить внимание на сетевые решения.

В то же время, уже существовала культура переносимых операционных сред, работающих в существенно гетерогенной сетевой среде, на машинах самой разной архитектуры: семейство Unix – подобных ОС. Важно, что относительно недорогие юникс – машины, часто работали вместе с серьёзными “монстрами”, мейнфреймами. Естественно, что и для них велась разработка подсистем графического пользовательского интерфейса и с самого начала стояла задача обеспечить взаимодействие различных машин в разнородной среде. Всеобщее при-

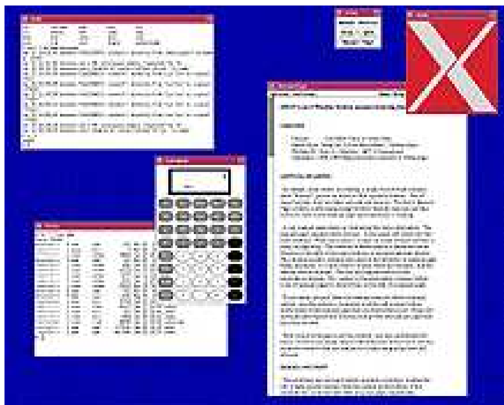

Рис. 2: классическая “визитка” X–Window

знание, в конце концов, получила разработка Массачусетского Технологического Института, “X1” (1984). В 1986 году была выпущена открытая для публики версия X10R3, которая была в скором времени портирована на рабочие станции Sun, Apollo и IBM-PC/AT и целый ряд более экзотического оборудования. Публичная версия системы получила название “X-Window” (рис. 2).

Как уже было сказано, для этой графической подсистемы. в первую голову, было важно обеспечить:

- переносимость на возможно больший ряд архитектур
- сетевую прозрачность: пользователь должен иметь возможность запустить программу (и возможно, не одну) на любой машине в сети, с любой архитектурой, при этом управляя её графическим интерфейсом со своей рабочей станции, на которой, в свою очередь, также работает графический интерфейс.

Из этих требований вытекает, что система

1. Должна быть модульной. Функции отрисовки графических примитивов должны быть, насколько возможно, простыми и полностью отделены от операционной системы.
2. Не может одновременно реализовывать графические функции и управление размещением на экране, эта функция должна быть возложена на отдельную программу — менеджер окон
3. Должна обеспечивать возможно простой протокол для работы в сети.

Соответственно, программирование такой системы — малоприятное занятие. Следует разбить все действия по отрисовке на большое количество примитивов, перевести графические команды в последовательность сетевых пакетов, правильным образом передать их в компьютерную сеть, да ещё и отслеживать при этом возникающие события. Более того, в основе концепции программирования Xlib, лежат функции обратного вызова, и вы обязаны самым внимательным образом относиться к обработке возникающих событий, заботится о правильности преобразования типов возвращаемых значений и т.д. Понятно, что отслеживать всю

эту механику вполне сравнимо с "закатом солнца вручную". Было принято решение, работать с X – Window (часто, используется сокращение по имени протокола — X11) через два промежуточных слоя. Первый — библиотека Xlib, обеспечивающая приемлемый для человека интерфейс, и в принципе – факультативный, а в реальности необходимый слой — так называемый графический тулкит — библиотеку, предоставляющую развитые возможности программирования графики.

Такая модульность приводит к тому, что у вас имеется не один, раз и навсегда заданный производителем набор системных вызовов управления графикой, а множество альтернатив: графических тулкитов, построенных на разных принципах, оказавшихся более или менее удачными, а часто тулкитов, базирующихся на предшественниках, используемых в цепочке — за 20 лет их была создано масса. Более того, так как все тулкиты базируются на ограниченном числе примитивов, имеющих аналоги во всех реализациях графических систем, но предоставляют пользователю высокоуровневые команды, граф. тулкит можно сконструировать многоплатформенным — написанная с его использованием программа, будет работать на Windows, Mac и т.д...

2.1 X – Window, стандартная граф. подсистема открытых систем

Как известно, история распорядилась так, что открытые и свободные системы начали развитие как попытка построения системы на идеях наиболее серьёзной из массовых систем того времени операционной среды — Unix (1983, GNU/Hurd, 1988, BSD*, 1990 (GNU)/Linux). Вполне понятно, что когда до этого дошло дело, за основу графической подсистемы была принята X-Window.

Стандартный современный пользователь Windows здесь. вероятно, скорчит кислую физиономию и спросит: "Ну и зачем мне это нужно? Пять минут назад, ты сам признавался, какое это малоприятное занятие — программировать иксы. В чём выгода? Я лучше уж погоняю что-нибудь директ-иксовое".

Более того, внимательный пользователь резонно заметит, что X всегда будет работать несколько медленнее, чем глубоко интегрированные в операционную систему решения, и будет прав. Пожалуй, лучший набор аргументов против X, можно найти в знаменитой книжке Гарфинкля "Unix Haters hanbook", IDG Books, 1994.

Но оглянитесь вокруг. Мы все работаем в сетевом окружении. И поверьте – возможность работать с приложением, запущенным в Университете, находясь в Академгородке; запускать счётную задачу с графическим интерфейсом на мощной и дорогой машине или тем более кластере — и в то же время, не расходовать их мощности на прорисовку графики — стоит многого. Более того, сетевая прозрачность "иксов" здорово облегчает объединение простых машин — посмотрите на собственную "локалку" — в вычислительный кластер, на котором можно решать самые, что ни на есть, бытовые задачи. Это работает. Альтернативные варианты — по разным причинам — либо не прижились, либо оказались хуже, либо много лет пребывают в состоянии вялого начала разработки. А замедление скорости, на современном "железе" — совсем незначительное.

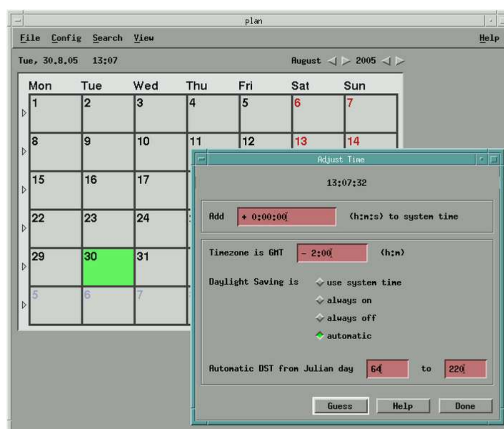

Рис. 3: Motif – приложения

3 Графические тулкиты

3.1 Беглый обзор. Что выбрать.

Итак, применение X-Window вполне осмыслено и весьма выгодно. Но чтобы пользоваться благами цивилизации, требуется развитый графический тулкит — вы же не хотите формировать сложные последовательности команд даже для самых простых операций? Что у нас имеется из этой оперы?

Исторически одним из первых (если не самым первым) тулкитом, ставшим индустриальным стандартом, стал Motif (1980, стандарт IEEE 1295, см рис. 3). Ага! Стандарт! Как Windows GDI?

Как бы не так. Ярчайший пример, как патентование и смежные лицензионные проблемы, плюс желание в обязательном порядке брать деньги и ни в коем случае не упускать ни единой копейки прямых продаж, превращает даже вполне успешных разработчиков в классический пример “собаки на сене”. Motif был типичным закрытым продуктом, что дало толчок созданию альтернативных тулкитов.

Не вдаваясь в историю, упомянем основные тулкиты, “выжившие” до настоящего времени на открытых системах:

- GTK (+)
- wxWidgets (wxWindows)
- Fox Toolkit
- Trolltech QT

GTK — Библиотека, первоначально нацеленная на предоставление базового набора примитивов при создании графического редактора GIMP. В настоящее время — полноценный тулкит. Реализована в объёктно – ориентированном стиле на обычном Си. Работает на X-Window и (базовые функции) — на Windows. LGPL.

Algorithm 1 скелет GTK приложения

```
main(...)
{
    gtk_init( &argc, &argv);
    window = gtk_window_new ( GTK_WINDOW_TOPLEVEL );
    gtk_signal_connect(GTK_OBJECT(window), ‘‘destroy’’, GTK_SIGNAL_FUNC(gtk_main_quit), NULL);
    gtk_widget_show( window );
    gtk_main();
   return (0);
}
```

wxWigdets — одна из известнейших библиотек. Многоплатформенная. Построена так, чтобы по-возможности, использовать родные средства среды. X-Window, Mac, Windows всех разновидностей, OS/2 (поддержка пока неустойчивая). LGPL

Fox Toolkit — многоплатформенный, потрясающе быстрый, удобный тулкит. X-Window, Mac (частично), Windows. LGPL.

QT — предмет сегодняшнего обзора. X-Window, Mac, Windows + встроенные устройства. GPL/Commercial.

Бегло взглянем на эти библиотеки, какая из них покажется более удобной?

⊗ Внешний вид приложений

3.2 Как с этим работают

Все упомянутые тулкиты, предполагают работу по хорошо знакомой Windows – программистам, классической схеме с циклом обработки событий: в точке входа в программу запускается бесконечный цикл, внутри которого производится отработка реакции на нажатие кнопок на клавиатуре, движения мышью и прочих **событий**. Внешние события — нажатия на клавиши клавиатуры, мыши, манипуляции с графическими элементами на экране, вызывают помещение в очередь некоторых кодов, на которые должен реагировать реализованный пользователем (программистом) обработчик. Общая схема одна и та же, различия в способе обработки событий. Наиболее существенное отличие от привычной для любителей windows схемы — вводится понятие widget (= window + gadget | object): каждое окно, в действительности является законченным объектом со своим, возможно сложным, поведением. Виджет может состоять из других виджетов. Все тулкиты оперируют с понятием виджета, как базой (и более того, виджеты лежат в основе даже низкоуровневого программирования через Xlib). Наиболее тесная аналогия виджету в терминах windows — “контрол”. Таким образом, можно забыть о хэндлах и функциях окна и работать в более высокоуровневом стиле (вспомните Windows GDI+ и Delphi !).

3.2.1 GTK

Структура приложения показана на фрагменте 1

Здесь передача сообщений происходит на основе ассоциаций с событиями связанных функций обратного вызова.

Algorithm 2 скелет wxWidgets приложения

```
class MyFrame: public wxFrame
{
    public:
        MyFrame (...);
        void OnQuit(wxCommandEvent& event);
    private:
         DECLARE_EVENT_TABLE()
};
// ------------------------------
enum { ID_Quit = 1 };
// ------------------------------
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(ID_Quit, MyFrame::OnQuit)
END_EVENT_TABLE()
// -----------------------------
IMPLEMENT_APP(MyApp); // главная точка входа в программу
```

Тулкит отличается обширным экстенсивным интерфейсом, который писался в разное время очень многими людьми и, судя по всему, не предназначен для освоения одним человеком. Gtk состоит из доброй дюжины надстроек и сервисов, таких как GLib, GdkPixbuf, GDK, GObject, Pango и т.д. Конечно, для создания простой формы пользователь не обязан знать все подробности, но все же такая "слоенка" вызывает некоторые опасения насчет стабильности и согласованного взаимодействия отдельных частей, особенно при быстрой разработке программы.

Документация: отвратительная.

Языковая поддержка (связки: basic, perl, python, schema, lisp), pascal (lazarus).

3.2.2 wxWidgets

Построена на C++, все специфические для программирования графического интерфейса функции, оформлены как методы класса. Пользовательская реакция на события, также должна быть методом класса. В то же время, для передачи событий, используются числовые коды (привет, Windows!). Оба подхода примиряются за счёт широкого использования специализированных макросов, скрывающих от пользователя истинную кухню работы с событиями (фрагм. 2)

Языковая поддержка (связки: perl, python, ruby)

3.2.3 Fox toolkit

Построена на C++, все специфические для программирования графического интерфейса функции, оформлены как методы класса. Пользовательская реакция на события, также должна быть методом класса. Также широко использует механизм макросов. Интересный механизм передачи событий: любое сообщение рассматривается как состоящее из двух частей, называемых селектором: комбинации типа сообщения и его идентификатора. При обработке сообщения, учитывается его селектор и отправитель. При совпадении селектора, вызыва-

ется зарегистрированный пользовательский метод. Это позволяет гибко вызывать методы реакции на события, обходя строгую типизацию.

Языковая поддержка (связки: python, ruby, schema)

3.3 QT

Наконец, главный герой сегодняшнего обзора и пожалуй, самая развитая из упомянутых, библиотека.

От ранее упомянутых (за исключением, разве что, wxWidgets) отличается не только идеологически, но и количественно: Qt — не только средство создания графического интерфейса, но и полный инструментарий для программирования (контейнеры, сетевые протоколы и протоколы ввода – вывода, XML, 2D и 3D графика, связка с OpenGL). Причём многие средства сконструированы так, чтобы быть совместимыми с решениями наиболее часто используемых в мире C++ библиотек (**STL**, **boost**).

Языковая поддержка (связки: perl, python, schema, lisp, C#)

Первоначально Qt разрабатывался Хаавардом Нордом (главный управляющий Trolltech) и Эриком Чамбенгом (президент компании Trolltech). Хаавард начал интересоваться проблемами создания графического интерфейса на C++ с 1988 года. Тогда он получил от Шведской компании заказ на разработку библиотеки, средствами которой можно было бы реализовать графический интерфейс приложений. Спустя пару лет, летом 1990 года, Хаавард и Эрик начали совместную работу над приложением баз данных, которое обрабатывало снимки, получаемые с аппарата ультразвукового обследования. Система должна была иметь возможность работы через графический интерфейс с пользователем, под управлением операционных систем Unix, Macintosh и Windows. Первый публичный выпуск датируется 1995 годом [1]. В качестве префикса, в именах классов, был выбран символ "Q", поскольку Хаавarду очень нравилось как он выглядел в Emacs. Символ "t" был выбран потому, что с него начиналось слово "toolkit", по аналогии с "Xt" – "X toolkit". Компания была зарегистрирована 4 марта 1994 года под названием "Quasar Technologies", которое затем было преобразовано в "Troll Tech", а затем и в "Trolltech". 20 мая 1995 года, Qt 0.90 была выложена на sunsite.unc.edu. Шесть дней спустя, выход библиотеки был анонсирован на comp.os.linux.announce. Это был первый публичный выпуск. Qt тогда могла работать как под управлением Windows, так и под управлением Unix, предоставляя разработчикам единый API (Прикладной Интерфейс). Библиотека была выпущена под двумя лицензиями: коммерческой – для разработки коммерческого программного обеспечения, и свободной – для разработки программ с открытым исходным кодом.

В апреле 1997 года вышла Qt 1.2. Решение Матиаса, об использовании Qt в качестве основы для KDE, де-факто сделало библиотеку стандартом, для разработки графического интерфейса в Linux. В сентябре 1997 года вышла Qt 1.3 [1].

В 2001 году вышла Qt 3. Теперь эта библиотека может работать под управлением Windows, Unix, Linux, Embedded Linux и Mac OS X. В ее состав вошли 42 новых класса, а общий объем кода перевалил за 500 000 строк. Qt 3 стала победительницей Software Development Times "Jolt Productivity Award" 2002 года. В 2001 году, библиотека официально вышла под лицензией GPL.

3.4 Общий обзор QT

3.4.1 Возможности[1]

Много–платформенность: Qt 3 доступна для **Windows** (любой версии), **Unix** (любой версии), **MacOS X** и **встроенных устройств**. Для запуска приложения на другой платформе, достаточно его лишь перекомпилировать.

Строгая иерархия классов Классы Qt построены по иерархическому принципу, для наследников **QObject** определено множество сервисных действий, включая сериализацию.

Элементы управления: Qt — это графический инструментарий, предлагающий богатый набор графических элементов управления: текстовые поля, раскрывающиеся списки, кнопки-переключатели,... Некоторые из них очень сложны. Например, список (listview) обладает возможностью многоколоночного представления с пиктограммами и кнопками-переключателями.

Классы поддержки графики и звука Перья, кисти, трансформации системы координат, двойная буферизация, поддержка растровых форматов, работы со шрифтами, анимация, поддержка печати, механизмы создания подсистемы помощи

Автоматическое размещение виджетов

XML и базы данных: Qt предлагает классы для работы с документами **XML** (используется Sax2 или DOM). Это простая, надежная и функциональная реализация.

Регулярные выражения

Классы шаблонов и контейнеров: Qt предлагает полезные классы для манипулирования списками, файлами, каталогами, строками, потоками,... Они более функциональны и удобны, чем аналогичные из STL.

Сетевой API: Qt предоставляет классы, облегчающие написание сетевых приложений: сокеты, DNS, FTP, HTTP, ...

API баз данных: Qt предоставляет классы для работы с базами данных: специальные виджеты, подключения к базам данных, SQL-запросы,...

OpenGL API: Qt предлагает классы для работы с библиотекой OpenGL.

Двухмерная графика (Canvas): Qt предлагает классы, оптимизированные для обработки быстро перемещающихся двухмерных объектов, известных как спрайты.

Стили: Можно полностью настроить внешний вид элементов управления. Qt может эмулировать стили всех известных инструментариев: Motiff, MFC, NextStep,...

[1] обобщающая таблица частично заимствована с [2]

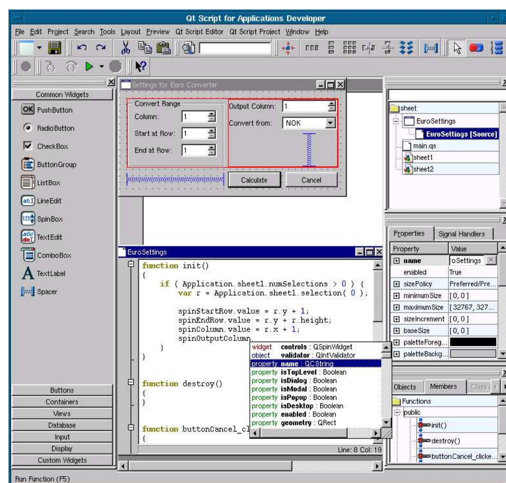

Рис. 4: QSA Designer

QSA: Существует скриптовый язык и автоматизированные (включая RAD, рис. 4) средства его встраивания [3]. QSA Designer — это RAD инструмент, который пользователи и разработчики используют для создания, редактирования, выполнения и отладки сценариев. QSA Designer включен в QSA-библиотеку и доступен всем приложениям. Кроме длинного названия QSA Designer включает средство создания GUI-интерфейса, аналогичное имеющемуся в Qt Designer, редактор кода Qt Script и интегрированный отладчик. Естественно, можно использовать и другие скриптовые языки, например, **Python**.

3.4.2 Реализация

Qt написан на C++ и предназначен, в первую очередь, для разработчиков на этом языке, хотя можно использовать и многие другие. Если говорить об ООП и графическом интерфейсе, то объектное программирование, контроль типов в духе C++ и встроенный в язык полиморфизм являются идеальным примером применения теории на практике. Не ограничиваясь стандартными средствами, QT идёт "немного дальше" классического C++, дополняя объекты информацией времени выполнения для дополнительных возможностей контроля типов.

3.4.3 Модель программирования

QT не вынуждает пользоваться какой-либо отдельной парадигмой, например, Model – View – Controller или Document – View. Можно использовать (или не использовать) любую. Для реализации библиотеки, использованы обычные механизмы C++, объектно-ориентированная архитектура была хорошо продумана. Поэтому инструментарий имеет согласованное именование, наследование, организацию классов и методов. Количество аргументов методов ограниченно только необходимыми. Их порядок постоянен для различных классов. А возвращаемое значение логично. И в тоже время сохраняется функциональность и простота.

Использовав однажды один из классов, Вы сможете использовать и другие, потому что они работают сходным образом. Так, чтобы в Qt получить элемент управления Edit, Вы должны создать объект QLineEdit с помощью оператора new, как в обычных классах C++. Тотчас же Вы получаете доступ ко всем методам объекта, независимо от того, отображен он или нет. Никаких специальных правил ожидания появления графического объекта в определённом состоянии, чтобы начать работать с ним, нет. Qt обладает различными средствами, упрощающими управление памятью. Объекты Qt автоматически уничтожаются при уничтожении их родителя. Многие классы обладают возможностью неявного совместного использования.

3.4.4 Локализация

Для локализации достаточно передавать внутри программы строки через функцию tr(). Это очень удобно при разработке — можно изменять строки непосредственно в коде. Специальная программа Qt Linguist извлечет все требующие перевода строки и в удобном виде их отобразит. Её интерфейс обеспечивает удобный перевод, возможность использования словаря, просмотр контекста строки, обнаружение конфликта клавиатурных сокращений, обнаружение новых непереведенных или измененных строк. Эта программа может использоваться переводчиком без познаний в области разработки программ. Перевод сохраняется в формате XML, поэтому он может быть легко использован в различных целях. Задача добавления к программе нового перевода заключается в создании нового файла с помощью Qt Linguist.

3.4.5 Инструментарий

Поддерживается полный набор интегрированных сред: IDE разработчика (KDeveloper), способная работать с широким набором языков программирования, QT3Designer — редактор форм (сравни с Delphi), QTLinguist — средство интернационализации. ⊗
Проект в KDeveloper — создание с нуля, импортирование. Стандартные и внешние способы сборки ⊗ KDeveloper

Создание формы в QT3Designer, компиляция в код. Использование C++ и Python. ⊗ Designer

3.5 Базовые механизмы QT[2]

3.5.1 Передача сообщений

Механизм делегатов Основная особенность библиотеки QT, первой бросающаяся в глаза — это механизм передачи сообщений, основанный на концепции делегатов (или что то же самое, механизме надёжной передачи сигналов типа сигнал – слот).

Этот механизм предоставляет вариант реализации функций обратного вызова, дополненный машинерией контроля типов. При возникновении в некотором объекте события, он может послать сигнал, ничуть не заботясь о том, будет ли эмитированный сигнал принят. Заинтересованный в реакции на событие объект, устанавливает ловушку на данный сигнал, по срабатыванию которой будет вызван указанный метод приёмника. Один сигнал может быть связан с неограниченным количеством приёмников, причём вместе с событием, передаются

[2] Информация взята из [4] и [5]

Algorithm 3 Описание класса, посылающего сигналы

```
class MY : public QObject
{ Q_OBJECT
    public:
        MY();
    signals:
        void SomeSignal ();
        void SomeCleverSignal (float arg, int j);
}
```

и связанные с ним данные (аргументы сигнала). Любой приёмник, может в произвольный момент отказаться от дальнейшего получения сигналов, цепочка распространения при этом нарушена не будет.

С точки зрения пользователя, для использования механизма сигнал-слотов нужно:

1. Унаследовать класс от базового **QObject**
2. Добавить в начало заголовка класса макрос **Q_OBJECT**
3. Описать эмитируемые сигналы в секции public **signals**. Сигналы описываются в точности, как обычные методы. (прг. 3)

посылка сигнала производится из любой функции приложения с использованием ключевого слова **emit** (компилятор C++ воспринимает emit как “пустоту”, и реальная обработка ложится на плечи *moc*):

```
void MY :: MyMethod() { emit SomeCleverSignal(3.14, 14); }
```

Если теперь какой – нибудь класс соединит один из своих методов – приёмников с данным сигналом при помощи функции connect[3], то при посылке сигнала, присоединённый метод будет вызван. Приёмники описываются в точности так же, как и сигналы, только вместо ключевого слова **signals:** используется ***slots:***. Тип функции приёмника, а также тип и её аргументов[4] должны в точности совпадать с объявленными в сигнале, это гарантирует правильность доставки сигнала.

Кроме прямой отправки и получения сигналов, широко используется трансляция сигналов, когда объект получатель при прибытии одного сигнала немедленно посылает другой. Для трансляции, новый сигнал просто указывается в функции connect.

Сигналы можно подсоединять (connect) и отсоединять (disconnect) в любой момент времени. Приёмников может быть множество, обработка сигналов производится **не** в порядке вызовов connect, а параллельно. При уничтожении объекта, все его сигнал – слотовые связи, автоматически разъединяются.

[3]

```
bool connect(QObject * sender, char * signal, QObject * reciver, char * slot);
```

[4] число аргументов приёмника может быть **меньше**, чем число аргументов сигнала

Попытка прямой, средствами C++, реализации подобного механизма, затруднена присущей C++ статической типизацией и недостатком информации о классах во время исполнения. Поэтому, Qt использует этап промежуточной компиляции при помощи метакомпилятора **moc**, на котором производятся необходимые для делегирования сигналов действия. *MOC* раскрывает макросы **Q_OBJECT**, **signals** и др., и генерирует реальный код.
Таким образом, компиляция Qt программы состоит из двух этапов: обработка файла компилятором moc и окончательная компиляция штатным C++ компилятором.

Работа с делегатами (слотами и сигналами), полностью доступна из QT3Designer.

⊗ Qt3Designer

Обработка событий При помощи механизма делегатов, в Qt доставляются сообщения о многих (практически, о всех) событиях, происходящих как в графических, так и классах общего назначения. Тем не менее, иногда этот механизм неудобен, особенно, если нужно централизовано обрабатывать множество схожих событий.

В этом случае предусматривается способ обработки сообщений посредством классической очереди. Ну, а чтобы сделать работу с очередью более удобной, вводится специальный класс QEvent, объекты которого могут посылаться объектам других классов функцией QApplication::notify. В объектах классов, которые должны обработать указанное событие, производится уточнение метода QObject::event. Класс может установить для себя фильтр, определяющий какие события будут обработаны функцией обработки, а какие будут игнорироваться. В любом случае, в организации огромных цепочек ***switch*** – ***case***, нет необходимости.

В классе QWidget предопределена масса функций для обработки стандартных событий, которые можно уточнить в соответствии с конкретными нуждами (клавиатура, мышь, действия с окном и фокусом ввода, реализация перетаскивания). Любой приёмник события может переслать событие (или сформировать своё) при помощи функций с префиксом **_qt**. Это позволяет отдельным элементам окон (виджетов) взаимодействовать друг с другом. Список предопределённых констант событий, смотрите в файле **q1xcompatibility.h**.

фильтрация событий Если нужно вмешаться в процесс обработки некоторых событий, нет нужды отказываться от использования QObject::event и писать специфический обработчик. Достаточно написать **фильтр события**, который выделит интересующие события из общего потока прямо в работе QObject::event. Интересно, что фильтровать события не обязательно в том же самом классе, который событие обрабатывает, это можно делать в любом другом и переключать фильтрацию динамически. Для этого используются методы QObject:: eventFilter(QObject *, QEvent *) — виртуальный метод, подлежащий уточнению; :: installEventFilter(QObject *), :: removeEventFilter(QObject *).

Например, если мне захочется отреагировать на помещение курсора мыши в окно, я могу установить фильтр:

```
bool MYCLASS :: eventFilter(QObject *pObj, QEvent *pEvt)
{
    if (Event_FocusIn == pEvt->type())
    {
      // ... обработать по вкусу, например, пролаять в колонки
```

```
        return true; // обработка закончена
     } else return false; // нет, обрабатывать дальше в QObject::event
  }
```

искусственное создание событий События можно генерировать искусственно, пользуясь методами QApplication::send|postEvent(QObject * Приёмник, QEvent *Событие)

3.5.2 Управление памятью

Qt самостоятельно управляет памятью и заботится о сборке мусора. Везде, где возможно, используется совместное разделение областей памяти вместо копирования. При программировании в Qt, все объекты должны создаваться динамически, вызовом new.

3.5.3 Навигация в иерархии

Все создаваемые при программировании в Qt объекты, занимают чёткое место в иерархии. Каждый объект знает своего предка. Благодаря этому, при уничтожении любого объекта (например, закрылось окно), все его потомки уничтожаются автоматически и рекурсивно, заботится о ручном освобождении памяти не нужно.

В любой момент, любой объект программы может проинспектировать своё место в иерархии и|или найти своего потомка вызовом методов parent() и child(). Каждый объект располагает расширеной информацией о себе и наследовании (метаобъектной информацией). Прямо во время исполнения, объект программы может получить информацию о имени своего класса (className()), проверить, от кого он унаследован (inherits(“”)) или принадлежность к конкретному классу (isA(“”)).

3.5.4 Поддержка многотредовости

Qt определяет класс QThread и полный набор примитивов синхронизации. Для запуска нового потока, достаточно наследоваться от QThread и желаемым образом переопределить метод run(). Поддержка многопоточности, тем не менее, не вполне полная: желательно, чтобы поток, изменяющий графические объекты, был только один (главный поток приложения). Если это не так, то безопаснее всего использовать для графических операций посылку нестандартных сообщений или, что менее безопасно, глобальную блокировку посредством QApplication::lock(). Кроме того, только главный поток имеет право обладать очередью сообщений. Механизм сигналов и слотов, может использоваться в потоках, но не может быть использован для взаимодействия между потоками, так как при выдаче сигнала слот исполняется в контексте потока передатчика, а не приёмника.

3.5.5 Поддержка создания пользовательских виджетов

⊗Designer

Коль скоро мы написали некоторый, возможно, сложный код, реализующий поведение окна (содержащего элемены управления или осуществляющего непростую визуализацию), было – бы неплохо “завернуть” этот код в единую подпрограмму, объект, библиотеку — называйте как хотите, для дальнейшего использования.

Это делается просто: с точки зрения идеологии, достаточно наследоваться от любого стандартного виджета, класса QWidget или его наследника, написать необходимый код поддержки и воспользоваться методами sizePolicy | setSizePolicy(), sizeHint(), чтобы указать правильные пределы изменения размеров.

С точки зрения практики, легче воспользоваться QT3Designer'ом и унаследоваться от полученой "рыбы".

⊗ QT3Designer

Чтобы новый виджет появился в дизайнере, можно либо добавить его на панель инструментов, либо написать плагин.

3.5.6 Особенности компиляции Qt приложений

1. Библиотека Qt не является частью стандартной библиотеки языка C++, поэтому она подключается к проекту при помощи опции линковщика **-l**.
2. Как правило, файлы Qt располагаются в своем обособленном каталоге, поэтому линковка должна осуществляться с опцией -L (путь к каталогу библиотеки).
3. Заголовочные файлы **qt** располагаются, как правило, в отдельном каталоге, который надо указывать при компиляции. Для этого используется опция **-I** компилятора.

Можно воспользоваться утилитой qmake, генерирующей стандартный мэйкфайл, либо автоматической системой сборки scons. Для сборки Qt программы scons'ом, достаточно указать компилятору определить макро "-DQT_THREAD_SUPPORT" (если требуется поддержка многотредовости), и заставить scons использовать инструмент "**qt**". Для этого можно написать в SConstruct:

```
defines = ["-D__LINUX__",  "-DQT_THREAD_SUPPORT"]
libs = ["#/libs"] # путь поиска для библиотек, собираемых по ходу дела
env = Environment(CCFLAGS = defines, LIBPATH = libs, LIBS = [’’доп. библиотеки, если есть’’]
, tools=[’default’, ’qt’])
Export(’env’)
```

После этого, можно просто указывать файлы .cpp, подлежащие сборки, и файлы .ui. генерируемые дизайнером. Метакомпилятор moc будет вызван автоматически, если это потребуется, равно как и весь последующий цикл сборки и компиляции программы будет выполнен автоматически.

Замечание о распространении текста

Авторское право на данный текст принадлежит Куликову Г.Э. Допускается свободное распространение, копирование и публикация любым способом данного текста в любых целях, при условии сохранения авторства. Авторское право на цитируемые материалы, принадлежит их авторам.

Список литературы

[1] Краткая история развития Qt http://www.opennet.ru/docs/RUS/qt3_prog/f71.html

[2] Сравнение MFC и QT http://qt.osdn.org.ua/mfc-vs-qt.html

[3] Быстрый, гибкий и забавный QT Script http://qt.osdn.org.ua/qsa.html

[4] QT Refrence Documentation, Free Edition (оффлайновый справочник в составе поставки Qt)

[5] Jasmin Blanchette, Mark Summerfield. Разработка графического интерфейса с помощью библиотеки Qt3. http://linuxcenter.ru/lib/books/qt3/